PySD Documentation

Release 0.7.10

James Houghton

Aug 11, 2017

Contents

Simulating System Dynamics Models in Python

This project is a simple library for running System Dynamics models in python, with the purpose of improving integration of Big Data and Machine Learning into the SD workflow.

PySD translates *Vensim* or *XMILE* model files into python modules, and provides methods to modify, simulate, and observe those translated models.

CHAPTER 1

Contents:

Installation

Installing via pip

To install the PySD package from the Python package index into an established Python environment, use the pip command:

```
pip install pysd
```

Installing from source

To install from the source, clone the project with git:

```
git clone https://github.com/JamesPHoughton/pysd.git
```

Or download the latest version from the project webpage: https://github.com/JamesPHoughton/pysd

In the source directory use the command

```
python setup.py install
```

Required Dependencies

PySD is built on python 2.7, and may not work as advertized on 3.x.

PySD calls on the core Python data analytics stack, and a third party parsing library:

- Numpy
- Scipy
- Pandas

- Matplotlib
- Parsimonious

These modules should build automatically if you are installing via *pip*. If you are building from the source code, or if pip fails to load them, they can be loaded with the same *pip* syntax as above.

Optional Dependencies

These Python libraries bring additional data analytics capabilities to the analysis of SD models:

- PyMC: a library for performing Markov chain Monte Carlo analysis
- Scikit-learn: a library for performing machine learning in Python
- NetworkX: a library for constructing networks
- GeoPandas: a library for manipulating geographic data

Additionally, the System Dynamics Translator utility developed by Robert Ward is useful for translating models from other system dynamics formats into the XMILE standard, to be read by PySD.

These modules can be installed using pip with syntax similar to the above.

Additional Resources

The PySD Cookbook contains a recipe on Installation_and_Setup that can help you get set up with both python and PySD.

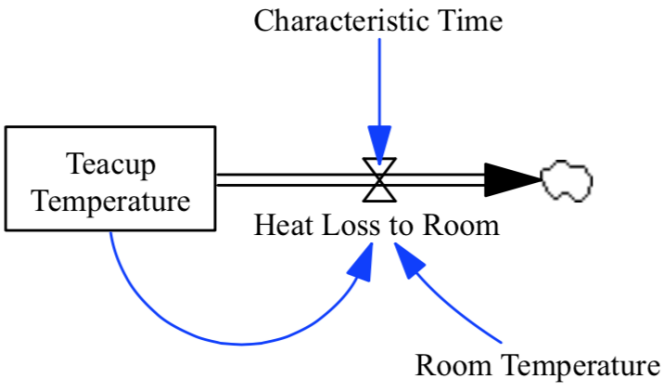
Basic Usage

Importing a model and getting started

To begin, we must first load the PySD module, and use it to import a supported model file:

```
import pysd
model = pysd.read_vensim('Teacup.mdl')
```

This code creates an instance of the PySD class loaded with an example model that we will use as the system dynamics equivalent of 'Hello World': a cup of tea cooling to room temperature.



To view a synopsis of the model equations and documentation, call the `doc()` method of the model class. This will generate a listing of all the model elements, their documentation, units, equations, and initial values, where appropriate. Here is a sample from the teacup model:

```
>>> print model.doc()
```

Running the Model

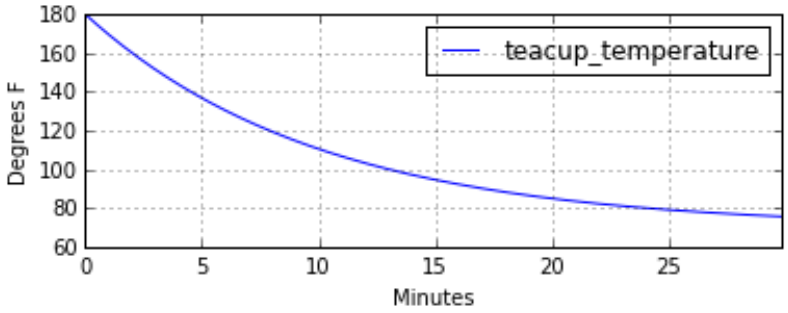
The simplest way to simulate the model is to use the `run()` command with no options. This runs the model with the default parameters supplied by the model file, and returns a Pandas dataframe of the values of the stocks at every timestamp:

```
>>> stocks = model.run()

t         teacup_temperature
0.000     180.000000
0.125     178.633556
0.250     177.284091
0.375     175.951387
...
```

Pandas gives us simple plotting capability, so we can see how the cup of tea behaves:

```
stocks.plot()
plt.ylabel('Degrees F')
plt.xlabel('Minutes')
```



Outputting various run information

The `run()` command has a few options that make it more useful. In many situations we want to access components of the model other than merely the stocks – we can specify which components of the model should be included in the returned dataframe by including them in a list that we pass to the `run()` command, using the return_columns keyword argument:

```
>>> model.run(return_columns=['Teacup Temperature', 'Room Temperature'])

t          Teacup Temperature    Room Temperature
0.000      180.000000            75.0
0.125      178.633556            75.0
0.250      177.284091            75.0
0.375      175.951387            75.0
...
```

If the measured data that we are comparing with our model comes in at irregular timestamps, we may want to sample the model at timestamps to match. The .run() function gives us this ability with the return_timestamps keyword argument:

```
>>> model.run(return_timestamps=[0,1,3,7,9.5,13.178,21,25,30])

t       Teacup Temperature
0.0     180.000000
1.0     169.532119
3.0     151.490002
7.0     124.624385
9.5     112.541515
...
```

Setting parameter values

In many cases, we want to modify the parameters of the model to investigate its behavior under different assumptions. There are several ways to do this in PySD, but the .run() function gives us a convenient method in the params keyword argument.

This argument expects a dictionary whose keys correspond to the components of the model. The associated values can either be a constant, or a Pandas series whose indices are timestamps and whose values are the values that the model component should take on at the corresponding time. For instance, in our model we can set the room temperature to a constant value:

```
model.run(params={'Room Temperature':20})
```

Alternately, if we believe the room temperature is changing over the course of the simulation, we can give the run function a set of time-series values in the form of a Pandas series, and PySD will linearly interpolate between the given values in the course of its integration:

```
import pandas as pd
temp = pd.Series(index=range(30), data=range(20,80,2))
model.run(params={'Room Temperature':temp})
```

Note that once parameters are set by the run command, they are permanently changed within the model. We can also change model parameters without running the model, using PySD's `set_components(params={})()` method, which takes the same params dictionary as the run function. We might choose to do this in situations where we'll be running the model many times, and only want to spend time setting the parameters once.

Setting simulation initial conditions

Finally, we can set the initial conditions of our model in several ways. So far, we've been using the default value for the initial_condition keyword argument, which is 'original'. This value runs the model from the initial conditions that were specified originally by the model file. We can alternately specify a tuple containing the start time and a dictionary of values for the system's stocks. Here we start the model with the tea at just above freezing:

```
model.run(initial_condition=(0, {'Teacup Temperature':33}))
```

Additionally we can run the model forward from its current position, by passing the initial_condition argument the keyword 'current'. After having run the model from time zero to thirty, we can ask the model to continue running forward for another chunk of time:

```
model.run(initial_condition='current',
          return_timestamps=range(31,45))
```

The integration picks up at the last value returned in the previous run condition, and returns values at the requested timestamps.

There are times when we may choose to overwrite a stock with a constant value (ie, for testing). To do this, we just use the params value, as before. Be careful not to use 'params' when you really mean to be setting the initial condition!

Querying current values

We can easily access the current value of a model component by calling its associated method (using python safe names) in the components subclass. For instance, to find the temperature of the teacup, we simply call:

```
model.components.teacup_temperature()
```

Supported functions

Vensim functions include:

Vensim	Python Translation
COS	np.cos
EXP	np.exp
MIN	min
<=	<=
STEP	functions.step
PULSE	functions.pulse
POISSON	np.random.poisson
EXPRND	np.random.exponential
SIN	np.sin
>=	>=
IF THEN ELSE	functions.if_then_else
LN	np.log
PULSE TRAIN	functions.pulse_train
RAMP	functions.ramp
INTEGER	int
TAN	np.tan
PI	np.pi
=	==
<	<
>	>
MODULO	np.mod
ARCSIN	np.arcsin
ABS	abs
^	**
LOGNORMAL	np.random.lognormal
MAX	max
SQRT	np.sqrt
ARCTAN	np.arctan
ARCCOS	np.arccos

Continued on next page

Table 1.1 – continued from previous page

Vensim	Python Translation
RANDOM NORMAL	self.functions.bounded_normal
RANDOM UNIFORM	np.random.rand
DELAY1	functions.Delay
DELAY3	functions.Delay
DELAY N	functions.Delay
SMOOTH3I	functions.Smooth
SMOOTH3	functions.Smooth
SMOOTH N	functions.Smooth
SMOOTH	functions.Smooth
INITIAL	functions.Initial
XIDZ	functions.XIDZ
ZIDZ	functions.XIDZ

np corresponds to the numpy package

Advanced Usage

The power of PySD, and its motivation for existence, is its ability to tie in to other models and analysis packages in the Python environment. In this section we'll discuss how those connections happen.

Replacing model components with more complex objects

In the last section we saw that a parameter could take on a single value, or a series of values over time, with PySD linearly interpolating between the supplied time-series values. Behind the scenes, PySD is translating that constant or time-series into a function that then goes on to replace the original component in the model. For instance, in the teacup example, the room temperature was originally a function defined through parsing the model file as something similar to:

```
def room_temperature():
    return 75
```

However, when we made the room temperature something that varied with time, PySD replaced this function with something like:

```
def room_temperature():
    return np.interp(t, series.index, series.values)
```

This drew on the internal state of the system, namely the time t, and the time-series data series that that we wanted to variable to represent. This process of substitution is available to the user, and we can replace functions ourselves, if we are careful.

Because PySD assumes that all components in a model are represented as functions taking no arguments, any component that we wish to modify must be replaced with a function taking no arguments. As the state of the system and all auxiliary or flow methods are public, our replacement function can call these methods as part of its internal structure.

In our teacup example, suppose we didn't know the functional form for calculating the heat lost to the room, but instead had a lot of data of teacup temperatures and heat flow rates. We could use a regression model (here a support vector regression from Scikit-Learn) in place of the analytic function:

```
from sklearn.svm import SVR
regression = SVR()
regression.fit(X_training, Y_training)
```

Once the regression model is fit, we write a wrapper function for its predict method that accesses the input components of the model and formats the prediction for PySD:

```
def new_heatflow_function():
    """ Replaces the original flowrate equation
        with a regression model"""
    tea_temp = model.components.teacup_temperature()
    room_temp = model.components.room_temperature()
    return regression.predict([room_temp, tea_temp])[0]
```

We can substitute this function directly for the heat_loss_to_room model component:

```
model.components.heat_loss_to_room = new_heatflow_function
```

Supplying additional arguments to the integrator

the `run()` function's argument *intg_kwargs* is a pass-through for keyword arguments to scipy's odeint function, and as such can take on any of the keywords that odeint recognizes.

User Functions Reference

These are the primary functions that control model import and execution.

Developer Documentation

About the Project

Motivation: The (coming of) age of Big Data

The last few years have witnessed a massive growth in the collection of social and business data, and a corresponding boom of interest in learning about the behavior of social and business systems using this data. The field of 'data science' is developing a host of new techniques for dealing with and analyzing data, responding to an increase in demand for insights and the increased power of computing resources.

So far, however, these new techniques are largely confined to variants of statistical summary, categorization, and inference; and if causal models are used, they are generally static in nature, ignoring the dynamic complexity and feedback structures of the systems in question. As the field of data science matures, there will be increasing demand for insights beyond those available through analysis unstructured by causal understanding. At that point data scientists may seek to add dynamic models of system structure to their toolbox.

The field of system dynamics has always been interested in learning about social systems, and specializes in understanding dynamic complexity. There is likewise a long tradition of incorporating various forms of data into system dynamics models.3 While system dynamics practice has much to gain from the emergence of new volumes of social data, the community has yet to benefit fully from the data science revolution.

There are a variety of reasons for this, the largest likely being that the two communities have yet to commingle to a great extent. A further, and ultimately more tractable reason is that the tools of system dynamics and the tools of data

analytics are not tightly integrated, making joint method analysis unwieldy. There is a rich problem space that depends upon the ability of these fields to support one another, and so there is a need for tools that help the two methodologies work together. PySD is designed to meet this need.

General approaches for integrating system dynamic models and data analytics

Before considering how system dynamics techniques can be used in data science applications, we should consider the variety of ways in which the system dynamics community has traditionally dealt with integration of data and models.

The first paradigm for using numerical data in support of modeling efforts is to import data into system dynamics modeling software. Algorithms for comparing models with data are built into the tool itself, and are usable through a graphical front-end interface as with model fitting in Vensim, or through a programming environment unique to the tool. When new techniques such as Markov chain Monte Carlo analysis become relevant to the system dynamics community, they are often brought into the SD tool.

This approach appropriately caters to system dynamics modelers who want to take advantage of well-established data science techniques without needing to learn a programming language, and extends the functionality of system dynamics to the basics of integrated model analysis.

A second category of tools uses a standard system dynamics tool as a computation engine for analysis performed in a coding environment. This is the approach taken by the Exploratory Modeling Analysis (EMA) Workbench6, or the Behavior Analysis and Testing Software (BATS)7. This first step towards bringing system dynamics to a more inclusive analysis environment enables many new types of model understanding, but imposes limits on the depth of interaction with models and the ability to scale simulation to support large analysis.

A third category of tools imports the models created by traditional tools to perform analyses independently of the original modeling tool. An example of this is SDM-Doc8, a model documentation tool, or Abdel-Gawad et. al.'s eigenvector analysis tool9. It is this third category to which PySD belongs.

The central paradigm of PySD is that it is more efficient to bring the mature capabilities of system dynamics into an environment in use for active development in data science, than to attempt to bring each new development in inference and machine learning into the system dynamics enclave.

PySD reads a model file – the product of a modeling program such as Vensim10 or Stella/iThink11 – and cross compiles it into Python, providing a simulation engine that can run these models natively in the Python environment. It is not a substitute for these tools, and cannot be used to replace a visual model construction environment.

Contributing to PySD

If you are interested in helping to develop PySD, the *PySD Development Pathway* lists areas that are ripe for contribution.

To get started, you can fork the repository and make contributions to your own version. When you're happy with your edits, submit a pull request to the main branch.

Development Tools

There are a number of tools that you might find helpful in development:

Test Suite

PySD uses the common model test suite found on github

You can run the test suite with the *test_pysd.py* module, which will execute each test and display some debugging info if there are issues.

The test runner will download the test suite, if it is not already installed, into the */tests/* directory. To update the test suite, you can run *get_tests.py* manually.

These tests run quickly and should be executed when any changes are made to ensure that current functionality remains intact.

The tes

Speed Tests

A set of speed tests are found in the *speed_test.py* module. These speed tests help understand how changes to the PySD module influence the speed of execution. These tests take a little longer to run than the basic test suite, but are not onerous. They should be run before any submission to the repository.

The speed test results are appended to 'speedtest_results.json', along with version and date information, so that before and after comparisons can be made.

The speed tests depend on the standard python `timeit` library.

Profiler

Profiling the code can help to identify bottlenecks in operation. To understand how changes to the code influence its speed, we should construct a profiling test that executes the PySD components in question. The file 'profile_pysd.py' gives an example for how this profiling can be conducted, and the file 'run_profiler.sh' executes the profiler and launches a view of the results that can be explored in the browser.

The profiler depends on `cProfile` and cprofilev

Python Linter

Pylint is a module that checks that your code meets proper python coding practices. It is helpful for making sure that the code will be easy for other people to read, and also is good fast feedback for improving your coding practice. The lint checker can be run for the entire packages, and for individual python modules or classes. It should be run at a local level (ie, on specific files) whenever changes are made, and globally before the package is committed. It doesn't need to be perfect, but we should aspire always to move in a positive direction.'

PySD Design Philosophy

Understanding that a focussed project is both more robust and maintainable, PySD aspires to the following philosophy:

- Do as little as possible.
- Anything that is not endemic to System Dynamics (such as plotting, integration, fitting, etc) should either be implemented using external tools, or omitted.
- Stick to SD. Let other disciplines (ABM, Discrete Event Simulation, etc) create their own tools.
- Use external model creation tools
- Use the core language of system dynamics.
- Limit implementation to the basic XMILE standard.
- Resist the urge to include everything that shows up in all vendors' tools.
- Emphasize ease of use. Let SD practitioners who haven't used python before understand the basics.

- Take advantage of general python constructions and best practices.
- Develop and use strong testing and profiling components. Share your work early. Find bugs early.
- Avoid firefighting or rushing to add features quickly. SD knows enough about short term thinking in software development to know where that path leads.

PySD Development Pathway

High priority features, bugs, and other elements of active effort are listed on the github issue tracker. To get involved see *Contributing to PySD*.

High Priority

- Subscripts/arrays Github Issue Track
- Refactor delays to take advantage of array architecture
- Improve translation of model documentation strings and units into python function docstrings

Medium Priority

- Outsource model translation to SDXchange model translation toolset
- Improve model exexution speed using cython, theano, numba, or another package
- Improve performance when returning non-stock model elements

Low Priority

- Import model component documentation in a way that enables doctest, to enable writing unit tests within the modeling environment.
- Handle simulating over timeseries
- Implement run memoization to improve speed of larger analyses
- Implement an interface for running the model over a range of conditions, build in intelligent parallelization.

Not Planned

- Model Construction
- Display of Model Diagrams
- Outputting models to XMILE or other formats

Ideas for Other Projects

- SD-lint checker (units, modeling conventions, bounds/limits, etc)
- Contribution to external Data Science tools to make them more appropriate for dynamic assistant

Current Features

- Basic XMILE and Vensim parser
- Established library structure and data formats
- Simulation using existing python integration tools
- Integration with basic python Data Science functionality
- Run-at-a-time parameter modification
- Time-variant exogenous inputs
- Extended backends for storing parameters and output values
- Demonstration of integration with Machine Learning/Monte Carlo/Statistical Methods
- Python methods for programmatically manipulating SD model structure
- Turn off and on 'traces' or records of the values of variables

Structure of the PySD module

PySD provides a set of translators that interpret a Vensim or XMILE format model into a Python native class. The model components object represents the state of the system, and contains methods that compute auxiliary and flow variables based upon the current state.

The components object is wrapped within a Python class that provides methods for modifying and executing the model. These three pieces constitute the core functionality of the PySD module, and allow it to interact with the Python data analytics stack.

Translation

The internal functions of the model translation components can be seen in the following documents

Vensim Translation

PySD parses a vensim '.mdl' file and translates the result into python, creating a new file in the same directory as the original. For example, the Vensim file `Teacup.mdl`:

becomes `Teacup.py`:

This allows model execution independent of the Vensim environment, which can be handy for deploying models as backends to other products, or for performing massively parallel distributed computation.

These translated model files are read by PySD, which provides methods for modifying or running the model and conveniently accessing simulation results.

Translated Functions

Ongoing development of the translator will support the full subset of Vensim functionality that has an equivalent in XMILE. The current release supports the following functionality:

Vensim	Python Translation
COS	np.cos

Continued on next page

Table 1.2 – continued from previous page

Vensim	Python Translation
EXP	np.exp
MIN	min
<=	<=
STEP	functions.step
PULSE	functions.pulse
POISSON	np.random.poisson
EXPRND	np.random.exponential
SIN	np.sin
>=	>=
IF THEN ELSE	functions.if_then_else
LN	np.log
PULSE TRAIN	functions.pulse_train
RAMP	functions.ramp
INTEGER	int
TAN	np.tan
PI	np.pi
=	==
<	<
>	>
MODULO	np.mod
ARCSIN	np.arcsin
ABS	abs
^	**
LOGNORMAL	np.random.lognormal
MAX	max
SQRT	np.sqrt
ARCTAN	np.arctan
ARCCOS	np.arccos
RANDOM NORMAL	self.functions.bounded_normal
RANDOM UNIFORM	np.random.rand
DELAY1	functions.Delay
DELAY3	functions.Delay
DELAY N	functions.Delay
SMOOTH3I	functions.Smooth
SMOOTH3	functions.Smooth
SMOOTH N	functions.Smooth
SMOOTH	functions.Smooth
INITIAL	functions.Initial
XIDZ	functions.XIDZ
ZIDZ	functions.XIDZ

np corresponds to the numpy package

Additionally, identifiers are currently limited to alphanumeric characters and the dollar sign $.

Future releases will include support for:

- subscripts
- arrays
- arbitrary identifiers

There are some constructs (such as tagging variables as 'suplementary') which are not currently parsed, and may throw an error. Future releases will handle this with more grace.

XMILE Translation

The XMILE reference documentation is located at:

- XMILE: http://www.iseesystems.com/community/support/XMILEv4.pdf
- SMILE: http://www.iseesystems.com/community/support/SMILEv4.pdf

The PySD module is capable of importing models from a Vensim model file (*.mdl) or an XMILE format xml file. Translation makes use of a Parsing Expression Grammar parser, using the third party Python library Parsimonious13 to construct an abstract syntax tree based upon the full model file (in the case of Vensim) or individual expressions (in the case of XMILE).

The translators then crawl the tree, using a dictionary to translate Vensim or Xmile syntax into its appropriate Python equivalent. The use of a translation dictionary for all syntactic and programmatic components prevents execution of arbitrary code from unverified model files, and ensures that we only translate commands that PySD is equipped to handle. Any unsupported model functionality should therefore be discovered at import, instead of at runtime.

The use of a one-to-one dictionary in translation means that the breadth of functionality is inherently limited. In the case where no direct Python equivalent is available, PySD provides a library of functions such as pulse, step, etc. that are specific to dynamic model behavior.

In addition to translating individual commands between Vensim/XMILE and Python, PySD reworks component identifiers to be Python-safe by replacing spaces with underscores. The translator allows source identifiers to make use of alphanumeric characters, spaces, or the $ symbol.

The model class

The translator constructs a Python class that represents the system dynamics model. The class maintains a dictionary representing the current values of each of the system stocks, and the current simulation time, making it a 'statefull' model in much the same way that the system itself has a specific state at any point in time.

The model class also contains a function for each of the model components, representing the essential model equations. The docstring for each function contains the model documentation and units as translated from the original model file. A query to any of the model functions will calculate and return its value according to the stored state of the system.

The model class maintains only a single state of the system in memory, meaning that all functions must obey the Markov property - that the future state of the system can be calculated entirely based upon its current state. In addition to simplifying integration, this requirement enables analyses that interact with the model at a step-by-step level. The downside to this design choice is that several components of Vensim or XMILE functionality – the most significant being the infinite order delay – are intentionally not supported. In many cases similar behavior can be approximated through other constructs.

Lastly, the model class provides a set of methods that are used to facilitate simulation. PySD uses the standard ordinary differential equations solver provided in the well-established Python library Scipy, which expects the state and its derivative to be represented as an ordered list. The model class provides the function .d_dt() that takes a state vector from the integrator and uses it to update the model state, and then calculates the derivative of each stock, returning them in a corresponding vector. A complementary function .state_vector() creates an ordered vector of states for use in initializing the integrator.

The PySD class

Internal Functions

This section documents the functions that are going on behaind the scenes, for the benefit of developers.

Special functions needed for model execution

These functions have no direct analog in the standard python data analytics stack, or require information about the internal state of the system beyond what is present in the function call. We provide them in a structure that makes it easy for the model elements to call.

Building the python model file

These elements are used by the translator to construct the model from the interpreted results. It is technically possible to use these functions to build a model from scratch. But - it would be rather error prone.

The PySD class provides the machinery to get the model moving, supply it with data, or modify its parameters. In addition, this class is the primary way that users interact with the PySD module.

The basic function for executing a model is appropriately named.run(). This function passes the model into scipy's odeint() ordinary differential equations solver. The scipy integrator is itself utilizing the lsoda integrator from the Fortran library odepack14, and so integration takes advantage of highly optimized low-level routines to improve speed. We use the model's timestep to set the maximum step size for the integrator's adaptive solver to ensure that the integrator properly accounts for discontinuities.

The .run() function returns to the user a Pandas dataframe representing the output of their simulation run. A variety of options allow the user to specify which components of the model they would like returned, and the timestamps at which they would like those measurements. Additional parameters make parameter changes to the model, modify its starting conditions, or specify how simulation results should be logged.

Complementary Projects

The most valuable component for better integrating models with *basically anything else* is a standard language for communicating the structure of those models. That language is XMILE. The draft specifications for this have been finalized and the standard should be approved in the next few months.

A python library for analyzing system dynamics models called the Exploratory Modeling and Analysis (EMA) Workbench is being developed by Erik Pruyt and Jan Kwakkel at TU Delft. This package implements a variety of analysis methods that are unique to dynamic models, and could work very tightly with PySD.

An excellent javascript library called sd.js created by Bobby Powers at SDlabs exists as a standalone SD engine, and provides a beautiful front end. This front end could be rendered as an iPython widget to facilitate display of SD models.

The Behavior Analysis and Testing Software(BATS) delveloped by Gönenç Yücel includes a really neat method for categorizing behavior modes and exploring parameter space to determine the boundaries between them.

CHAPTER 2

Additional Resources

PySD Cookbook

A cookbook of simple recipes for advanced data analytics using PySD is available at: http://pysd-cookbook.readthedocs.org/

The cookbook includes models, sample data, and code in the form of ipython notebooks that demonstrate a variety of data integration and analysis tasks. These models can be executed on your local machine, and modified to suit your particular analysis requirements.

Contributing

The code for this package is available at: https://github.com/JamesPHoughton/pysd

If you find a bug, or are interested in a particular feature, see the project's issue tracker on github.

If you are interested in contributing to the development of PySD, see the Developer Documentation listed above, create a fork on github, and submit your pull requests when ready.

Support

For additional help or consulting, contact james.p.houghton@gmail.com

CHAPTER 3

Indices and tables

- genindex